Static mapping of real-time applications onto massively parallel processor arrays

Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert de Simone

Static mapping of real-time applications onto massively parallel processor arrays

Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert de Simone
INRIA, France
Zhen Zhang
IRT SystemX/FSF, France

***Abstract*—On-chip networks (NoCs) used in multiprocessor systems-on-chips (MPSoCs) pose significant challenges to both on-line (dynamic) and off-line (static) real-time scheduling approaches. They have large numbers of potential contention points, have limited internal buffering capabilities, and network control operates at the scale of small data packets. Therefore, efficient resource allocation requires requires scalable algorithms working on hardware models with a level of detail that is unprecedented in real-time scheduling. We consider here a static scheduling approach, and we target massively parallel processor arrays (MPPAs), which are MPSoCs with large numbers (hundreds) of processing cores. We first identify and compare the hardware mechanisms supporting precise timing analysis and efficient resource allocation in existing MPPA platforms. We determine that the NoC should ideally provide the means of enforcing a global communications schedule that is computed off-line (before execution) and which is synchronized with the scheduling of computations on processors. On the software side, we propose a novel allocation and scheduling method capable of synthesizing such global computation and communication schedules covering all the execution, communication, and memory resources in an MPPA. To allow an efficient use of the hardware resources, our method takes into account the specificities of MPPA hardware and implements advanced scheduling techniques such as software pipelining and pre-computed preemption of data transmissions. We evaluate our technique by mapping two signal processing applications, for which we obtain good latency, throughput, and resource use figures.**

I. Introduction

One crucial problem in real-time scheduling is that of ensuring that the application software and the implementation platform satisfy the hypotheses allowing the application of specific *schedulability analysis* techniques [29]. Such hypotheses are the availability of a priority-driven scheduler or the possibility of including scheduler-related costs in the durations of the tasks. Classical work on real-time scheduling [14], [22], [19] has proposed formal models allowing schedulability analysis in classical mono-processor and distributed settings. But the advent of *multiprocessor systems-on-chip (MPSoC)* architectures imposes significant changes to these models and to the scheduling techniques using them.

MPSoCs are becoming prevalent in both general-purpose and embedded systems. Their adoption is driven by scalable performance arguments (concerning speed, power, *etc.*), but this scalability comes at the price of increased complexity of both the software and the software mapping (allocation and scheduling) process.

Part of this complexity can be attributed to the steady increase in the *quantity* of software that is run by a single system. But there are also significant *qualitative* changes concerning both the software and the hardware. In software, more and more applications include *parallel* versions of classical signal or image processing algorithms [43], [4], [21], which are best modeled using data-flow models (as opposed to *independent tasks*). Providing functional and real-time correctness guarantees for parallel code requires an accurate control of the interferences due to concurrent use of communication resources. Depending on the hardware and software architecture, this can be very difficult [44], [27].

Significant changes also concern the execution platforms, where the gains predicted by Moore's law no longer translate into improved single-processor performance, but in a rapid increase of the number of processor cores placed on a single chip [10]. This trend is best illustrated by the *massively parallel processor arrays (MPPAs)*, which we target in this paper. MPPAs are MPSoCs characterized by:

- Large numbers of processing cores, ranging in current silicon implementations to a few tens to a few hundreds [42], [33], [1], [17]. The cores are typically chosen for their area or energy efficiency instead of raw computing power.
- A regular internal structure where processor cores and internal storage (RAM banks) are divided among a set of identical *tiles*, which are connected through one or more NoCs with regular structure (*e.g.* torus, mesh).

Industrial [42], [33], [1], [17] and academic [23], [9], [40] MPPA architectures targeting hard real-time applications already exist, but the problem of mapping applications on them remains largely open. There are two main reasons to this. The first one concerns the NoCs: as the tasks are more tightly coupled and the number of resources in the system increases, the on-chip networks become critical resources, which need to be explicitly considered and managed during real-time scheduling. Recent work [40], [28], [35] has determined that NoCs have distinctive traits requiring significant changes to classical real-time scheduling theory [22]. The second reason concerns automation: the complexity of MPPAs and of the (parallel) applications mapped on them is such that the *allocation and scheduling must be largely automated*.

We address here this automation need, and unlike previous work on the subject we focus on static (off-line) scheduling approaches. In theory, off-line algorithms allow the computation of *scheduling tables* specifying an *optimal* allocation

and real-time scheduling of the various computations and communications onto the resources of the MPPA. In practice, this ability is severely limited by 3 factors:

1) The **application** may exhibit a high degree of dynamicity due to either environment constraints or to execution time variability resulting from data-dependent conditional control.[1]
2) The **hardware** may not allow the implementation of optimal scheduling tables. For instance, most MPPA architectures provide only limited control over the scheduling of communications inside the NoC.
3) The **mapping** problems we consider are NP-hard. In practice, this means that optimality cannot be attained, and that efficient heuristics are needed.

In the remainder of the paper we assume that our applications are static enough to benefit from off-line scheduling (*e.g.* parallelized versions of signal processing and embedded control algorithms).

Our contributions concern the hardware and the mapping. We start with an in-depth review of MPPA/NoC architectures with support for real-time scheduling. This analysis allows us to determine that NoCs allowing static communication scheduling offer the best support to off-line application mapping. This analysis also allows us to define a new technique and tool, called LoPhT, for automatic real-time mapping and code generation, whose global flow is pictured in Fig. 1. Our tool takes as input data-flow synchronous specifications

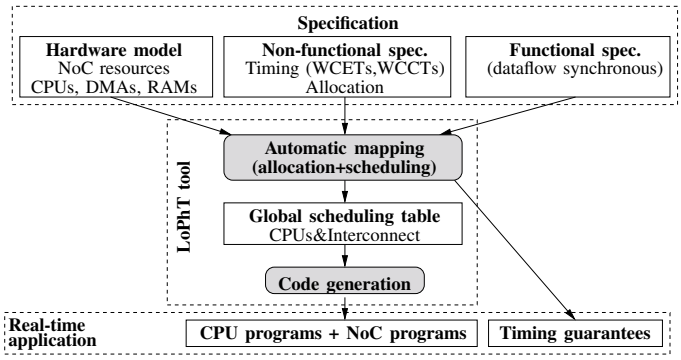

Fig. 1. Global flow of the proposed mapping technique

and precise hardware descriptions including all potential NoC contention points. It uses advanced off-line scheduling techniques such as software pipelining and pre-computed preemption, and it takes into account the specificities of the MPPA hardware to build *scheduling tables* that provide good latency and throughput guarantees and ensure an efficient use of computation and communication resources. Scheduling tables are then automatically converted into sequential code ensuring the correct ordering of operations on each resource and the respect of the real-time guarantees. We provide results for two applications that are mapped onto the MPPA platform described in [15]. Our results show that the off-line mapping of communications not only allows us to provide static latency and throughput guarantees, but may also improve the speed of the application.

[1]Implementing an optimal control scheme for such an application may require more resources than the application itself, which is why dynamic/on-line scheduling techniques are often preferred.

II. MPPA/NoC architectures for the real-time

We start with a general introduction to MPPA platforms, and then present the main characteristics of existing NoCs. We are mainly interested here in the traffic management mechanisms supporting real-time implementation.

A. Structure of an MPPA

Our work concerns hard real-time systems where timing guarantees must be determined by static analysis methods before system execution. Complex memory hierarchies involving multiple cache levels and cache coherency mechanisms are known to complicate timing analysis [44], [25], and we assume they are not used in the MPPA platforms we consider. Under this hypothesis, *all* data transfers between tiles are performed through one or more NoCs.

A NoC can be described in terms of point-to-point communication links and NoC routers which perform the routing and scheduling (arbitration) functions. Fig. 2 provides the description of a 2-dimensional (2D) mesh NoC like the ones used in the Adapteva Epiphany [1], Tilera TilePro[42], or DSPIN[36]. The structure of a router in a 2D mesh NoC is

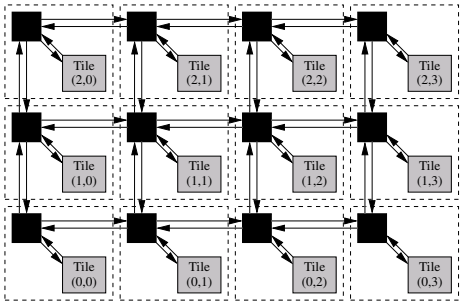

Fig. 2. A 4x3 tile MPPA platform with 2D mesh NoC. Black rectanges are the NoC routers.

described in Fig. 3. It has 5 connections (labeled North, South, West, East, and Local) to the 4 routers next to it and to the local tile. Each connection is formed of a routing component, which we call *demultiplexer* (labeled **D** in Fig. 3), and a scheduling/arbitration component, which we call *multiplexer* (labeled **M**). Data enters the router through demultiplexers and exits through multiplexers.

To allow on-chip implementation at a reasonable area cost, NoCs use very simple routing algorithms. For instance, the Adapteva [1], Tilera [42], and DSPIN NoCs [36] use an X-first routing algorithm where data first travels all the way in the X direction, and only then in the Y direction. Furthermore, all NoCs mentioned in this paper use simple *wormhole* switching approaches [34] requiring that all data of a communication unit (*e.g.* packet) follow the same route, in order, and that the communication unit is not logically split during transmission.

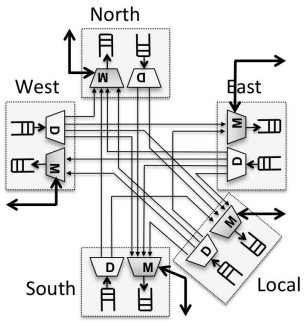

Fig. 3. Generic router for a 2D mesh NoC with X-first routing policy

The use of a wormhole switching approach is justified by the limited buffering capabilities of NoCs [35] and by the possibility of decreasing transmission latencies (by comparison with more classical store-and-forward approaches). But the use of wormhole switching means that one data transmission unit (such as a packet) is seldom stored in a single router buffer. Instead, a packet usually spans over several routers, so that its *transmission strongly synchronizes multiplexers and demultiplexers along its path*.

B. Support for real-time implementation

Given the large number of potential contention points (router multiplexers), and the synchronizations induced by data transmissions, providing *tight* static timing guarantees is only possible if some form of flow control mechanism is used.

In NoCs based on *circuit switching* [3], inter-tile communications are performed through dedicated *communication channels* formed of point-to-point physical links. Two channels cannot share a physical link. This is achieved by statically fixing the output direction of each demultiplexer and the data source of each multiplexer along the channel path. Timing interferences between channels are impossible, which radically simplifies timing analysis, and the latency of communications is low. But the absence of resource sharing is also the main drawback of circuit switching, resulting in low numbers of possible communication channels and low utilization of the NoC resources. Reconfiguration is usually possible, but it carries a large timing penalty.

Virtual circuit switching is an evolution of circuit switching which allows resource sharing between circuits. But resource sharing implies the need for arbitration mechanisms inside NoC multiplexers. Very interesting from the point of view of timing predictability are NoCs where arbitration is based on time division multiplexing (TDM), such as Aethereal [23], Nostrum [31], and others [41]. In a TDM NoC, all routers share a common time base. The point-to-point links are reserved for the use of the virtual circuits following a fixed cyclic schedule (a scheduling table). The reservations made on the various links ensure that communications can follow their path without waiting. TDM-based NoCs allow the computation of *precise latency and throughput guarantees*. They also ensure a strong *temporal isolation* between virtual circuits, so that changes to a virtual circuit do not modify the real-time characteristics of the other.

When no global time base exists, the same type of latency and throughput guarantees can be obtained in NoCs relying on *bandwidth management* mechanisms such as Kalray MPPA [33], [26]. The idea here is to ensure that the throughput of each virtual circuit is limited to a fraction of the transmission capacity of a physical point-to-point link, by either the emitting tile or by the NoC routers. Two or more virtual circuits can share a point-to-point link if their combined transmission needs are less than what the physical link provides.

But TDM and bandwidth management NoCs have certain limitations: One of them is that latency and throughput are correlated [40], which may result in a waste of resources. But the latency-throughput correlation is just one consequence of a more profound limitation: TDM and bandwith management NoCs largely ignore the fact that the needs of an application may change during execution, depending on its *state*. For instance, when scheduling a data-flow synchronous graph with the objective of reducing the duration of one computation cycle (also known as *makespan* or *latency*), it is often useful to allow some communications to use 100% of the physical link, so that they complete faster, before allowing all other communications to be performed.

One way of taking into account the application state is by using NoCs with support for *priority-based scheduling* [40], [36], [17]. In these NoCs, each data packet is assigned a priority level (a small integer), and NoC routers allow higher-priority packets to pass before lower-priority packets. To avoid *priority inversion* phenomenons, higher-priority packets have the right to preempt the transmission of lower-priority ones. In turn, this requires the use of one separate buffer for each priority level in each router multiplexer, a mechanism known as *virtual channels (VCs)* in the NoC community[36].

The need for VCs is the main limiting factor of priority-based arbitration in NoCs. Indeed, adding a VC is as complex as adding a whole new NoC[46], [11], and NoC resources (especially buffers) are expensive in both power consumption and area [32]. To our best knowledge, among existing silicon implementations only the Intel SCC chip offers a relatively large numbers of VCs (eight) [17], and it is targeted at high-performance computing applications. Industrial MPPA chips targeting an embedded market usually feature multiple, specialized NoCs [42], [1], [26] without virtual channels. Other NoC architectures feature low numbers of VCs. Current research on priority-based communication scheduling has already integrated this limitation, by investigating the sharing of priority levels [40].

Significant work already exists on the mapping of real-time applications onto priority-based NoCs [40], [39], [35], [28]. This work has shown that priority-based NoCs support the efficient mapping of *independent* tasks.

But we already explained that the large number of computing cores in an MPPA means that applications are also likely to include parallelized code which is best modeled by large sets of relatively small dependent tasks (data-flow graphs) with predictable functional and temporal behavior [43], [4], [21]. Such timing-predictable specifications are those that

can *a priori* take advantage of a static scheduling approach, which provides best results on architectures with support for static communication scheduling [42], [18], [15]. Such architectures allow the construction of an efficient (possibly optimal) global computation and communication schedule, represented with a scheduling table and implemented as a set of synchronized *sequential computation and communication programs*. Computation programs run on processor cores to sequence task executions and the initiation of communications. Communication programs run on specially-designed micro-controllers that control each NoC multiplexer to fix the order in which individual data packets are transmitted. Synchronization between the programs is ensured by the data packet communications themselves.

Like in TDM NoCs, the use of global computation and communication scheduling tables allows the computation of very precise latency and throughput estimations. Unlike in TDM NoCs, NoC resource reservations can depend on the application state. Global time synchronization is not needed, and existing NoCs based on static communication scheduling do not use it[42], [18], [15]. Instead, global synchronization is realized by the data transmissions (which eliminates some of the run-time pessimism of TDM-based approaches).

The microcontrollers that drive each NoC router multiplexer are similar in structure to those used in TDM NoCs to enforce the TDM reservation pattern. The main difference is that the communication programs are usually longer than the TDM configurations, because they must cover longer execution patterns. This requires the use of larger program memory (which can be seen as part of the tile program memory[15]). But like in TDM NoCs, buffering needs are limited and no virtual channel mechanism is needed, which results in lower power consumption.

From a mapping-oriented point of view, determining exact packet transmission orders cannot be separated from the larger problem of building a global scheduling table comprising both computations and communications. By comparison, mapping onto MPPAs with TDM-based or bandwith reservation-based NoCs usually separates task allocation and scheduling from the synthesis of a NoC configuration independent from the application state [30], [4], [47] .

Under static communication scheduling, there is little run-time flexibility, as all scheduling possibilities must be considered during the off-line construction of the global scheduling table. For very dynamic applications this can be difficult. This is why existing MPPA architectures that allow static communication scheduling also allow communications with dynamic (Round Robin) arbitration.

In conclusion, NoCs allowing static communication scheduling offer the best temporal precision in the off-line mapping of dependent tasks (data-flow graphs), while priority-based NoCs are better at dealing with more dynamic applications. As future systems will include both statically parallelized code and more dynamic aspects, NoCs will include mechanisms supporting both off-line and on-line communication scheduling. Significant work already exists on the real-time mapping for priority-based platforms, while little work has addressed the NoCs with static communication scheduling. In the remainder of the paper we address this issue by proposing a new off-line scheduling technique.

III. Related work

The previous section already discussed existing work on taking into account NoC structure during real-time mapping.

But our work is also closely related to classical results on the off-line real-time mapping of dependent task systems onto multiprocessor and distributed architectures [19], [45], [16], [24]. The objective is the same as in our case: the synthesis of optimized time-triggered execution patterns. But there are also significant differences. Most NoCs rely on wormhole routing, which requires synchronized reservation of the resources along the communication paths. By comparison, the cited papers either use a store-and-forward routing paradigm [19], [24], [16] that is inapplicable to NoCs, or simply do not model the communication media [45].

A second difference concerns the complexity of the architecture description. MPPAs have more computation and communication resources than typical distributed architectures considered in classical real-time. Moreover, resource characterization has clock cycle precision. To scale up without losing timing precision, we employ scheduling heuristics of low computational complexity, avoiding the use of techniques such as backtracking [19], but taking advantage of low-level architectural detail concerning the NoC.

A third difference concerns fine-grain resource allocation and scheduling operations, such as the allocation of data variables into memory banks or the scheduling of DMA commands onto the processor cores. These operations are often overlooked in classical distributed scheduling. But on an MPPA platform they can have a major impact on execution durations. This why we need to explicitly consider them at off-line mapping time.

More generally, our work is also close to previous results on static, but not real-time, mapping of applications onto MPPAs. The StreamIt compiler [2] uses scheduling tables as an internal compiler representation. But StreamIt uses timing information coming from simulations to guide the mapping process, and does not take into account timing interferences due to the mapping itself. Previous work on the compilation of the SigmaC language onto the Kalray MPPA platform [4] also uses timing information coming from simulations, but its objective is to generate code for a micro-kernel employing dynamic task scheduling.

Similar approaches are taken in more dynamic techniques aimed at signal processing systems [8], [20], [7]. Techniques such as DOL[5] or the one of Zhai *et al.* [47] allow for real-time mapping, but without considering the details of the NoC. There is a price to pay for this: In DOL, mapping results are not proved safe, because timing information comes from simulations which may not cover the worst case. In the approach of Zhai *et al.*, NoC resource allocation that guarantees the latency of inter-tile communications is done

before application mapping, and is not optimized afterwards. Our method of taking into account low-level NoC details could potentially be used to improve such mapping techniques.

IV. Hardware model

A. MPPA hardware

We use in this paper the MPPA architecture of Djemal *et al.* [15]. The overall architecture of the MPPA and of its routers is that of Figures 2 and 3. NoC control operates at *packet* level, each packet being a bounded sequence of *flits* (FLow control unITs, the data unit that can be sent over a NoC link in one clock cycle). NoC routing if of X-first type.

Packet scheduling (arbitration) in the router multiplexers can be either dynamic (with a Round Robin policy), or fixed off-line. In the second case, the order in which packets are transmitted is specified with a communication program stored in the router controller component of Fig. 4. Regardless of the arbitration type, the transmission of a packet can never be interrupted. The NoC only allows the buffering of 3 flits for each link.

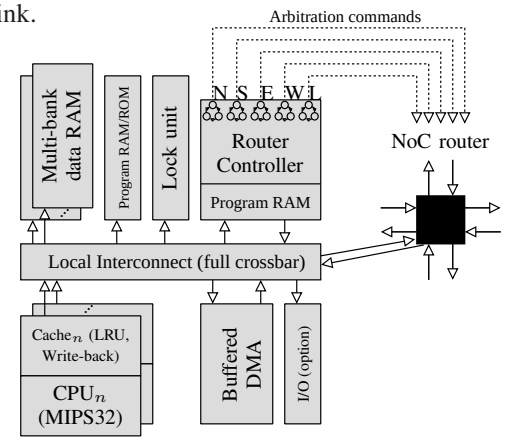

Fig. 4. The computing tile of our MPPA architecture

For efficiency and timing predictability purposes, the tiles of our MPPA are formed of the following components (Fig. 4):

- 16 MIPS32 processor cores with separate instruction and data caches.
- 32 data memory banks that can be accesses independently, for a total of maximum 2 Mbytes of RAM. The program is stored in a separate RAM or ROM bank.
- A DMA unit which allows the queuing of multiple DMA commands in order to diminish timing interference between computations and communications.
- A hardware lock component. The locks allow low-overhead synchronization and make it easier to preserve temporal predictability because they do not alter the cache state (as opposed to interrupt-based synchronization). Locks are used to ensure exclusive access to data RAM banks.
- A NoC router controller that runs the arbitration programs of the NoC router.

All these components are linked together through a full crossbar *local interconnect* which is also linked to the NoC.

B. Resource modeling

To allow off-line mapping onto our architectures, we need to identify the set of *abstract* computation and communication resources that are considered during allocation and scheduling.

We associate one communication resource to each of the multiplexers of NoC routers and to each DMA. We name them as follows: $N(i,j)(k,l)$ is the inter-router wire going from $Tile(i,j)$ to $Tile(k,l)$; $In(i,j)$ is the output of the router (i,j) to the local tile; $DMA(i,j)$ is the output of $Tile(i,j)$ to the local router.

This paper focuses on NoC modeling and handling. To this end, we consider in this paper a resource model that simplifies as much as possible the representation of the computing tiles. All the 16 processor cores of the tile are seen as a single, very fast computing resource. This means that operations will be allocated to the tile as if it were a sequential processor, but the allocated operations are in fact parallel code running on all 16 processors. In Fig. 2 there are just 12 tile resources representing 192 processor cores. This simplification largely reduces the complexity of our presentation, and also satisfies our evaluation needs, given that the 2 applications used as examples can be organized into operations that are easily parallelized onto 16 processors.

C. Communication durations

All inter-tile data transmissions are performed using the DMA units. If a transmission is not blocked on the NoC, then its duration on the sender side only depends on the size of the transmitted data. The exact formula is $d = s + \lceil s/\mathit{maxpayload} \rceil * \mathit{PacketHeaderSize}$, where d is the duration in clock cycles of the DMA transfer from the start of the transmission to the cycle where a new transmission can start, s is the data size in 32-bit words, *MaxPayload* is the maximum payload of a NoC packet produced by the DMA (in 32-bit words), and *PacketHeaderSize* is the number of cycles that are lost for each packet in the chosen NoC. In our case, *MaxPayload*=16 flits and *PacketHeaderSize*=4 flits.

In addition to this transmission duration, we must also account in our computations for:

- The DMA transfer initiation, which consists in 3 uncached RAM accesses plus the duration of the DMA reading the payload of the first packet from the data RAM. This cost is over-approximated as 30 cycles.
- The latency of the NoC, which is the time needed for one flit to traverse the path from source to destination. This latency is of $3 * n$, where n is the number of NoC multiplexers on the route of the transmission.

V. Application specification

The application to be mapped is specified under the form of a data-flow graph (known as a dependent task system in the real-time community). The input of our tool is a full-fledged data-flow synchronous formalism like that of [24]. For presentation reasons, however, we simplify it as follows.

A data-flow graph D is a directed graph with two types of arcs $D = \{T(D), A(D), \Delta(D)\}$. Here, $T(D)$ is the finite

set of tasks (data-flow blocks). The finite set $A(D)$ contains dependencies of the form $a = (src(a), dst(a), type(a))$, where $src(a), dst(a) \in T(D)$ are the source, respectively the destination task of a, and $type(a)$ is the type of data transmitted from $src(a)$ to $dst(a)$. The directed graph determined by $A(D)$ must be acyclic. The finite set $\Delta(D)$ contains *delayed* dependencies of the form $\delta = (src(\delta), dst(\delta), type(\delta), depth(\delta))$, where $src(\delta)$, $dst(\delta)$, $type(\delta)$ have the same meaning as for simple dependencies and $depth(\delta)$ is a strictly positive integer called the *depth* of the dependency.

Data-flow graphs have a cyclic execution model. At each execution cycle of the task set, each of the tasks is executed exactly once. We denote with t^n the instance of task $t \in T(D)$ for cycle n. The execution of the tasks inside a cycle is partially ordered by the dependencies of $A(D)$. If $a \in A(D)$ then the execution of $src(a)^n$ must be finished before the start of $dst(a)^n$, for all n.

The dependencies of $\Delta(D)$ impose an order between tasks of successive execution cycles. If $\delta \in \Delta(D)$ then the execution of $src(\delta)^n$ must complete before the start of $dst(\delta)^{n+depth(\delta)}$, for all n.

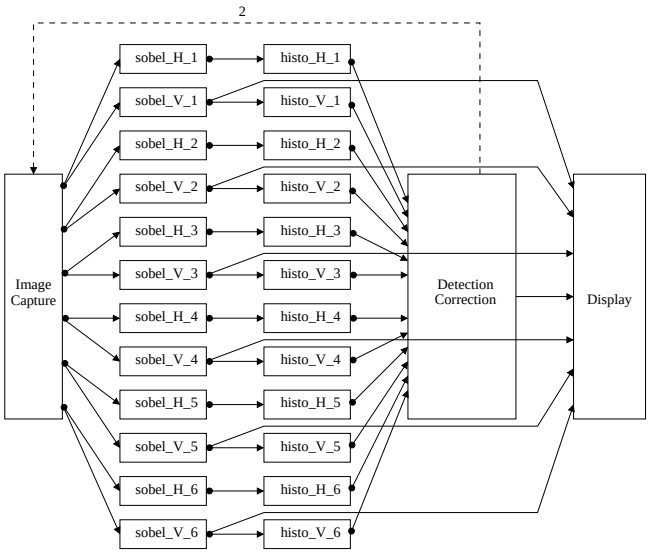

Fig. 5. Data-flow graph of a platooning application

We exemplify our formalism with the data-flow graph of Fig. 5, which is a simplified version of an automotive platooning application [37]. In our figure, each block is a task, solid arcs are simple dependencies, and the dashed arc is a delayed dependency of depth 2. The application is run by a car to determine the position (distance and angle) of another car moving in front of it. It works by cyclically capturing an input image of fixed size. This image is passed through an edge-detecting Sobel filter and then through a histogram search to detect dominant edges (horizontal and vertical). This information is used by the detection and correction function to determine the position of the front car. The whole process is monitored on a display. The delayed dependency represents a feedback from the detection and correction function that allows the adjustment of image capture parameters.

The Sobel filter and the histogram search are parallelized. Each of the *Sobel_H* and *Sobel_V* functions receives one sixth of the whole image (a horizontal slice).

A. Non-functional specification

For each task $\tau \in T(D)$, we define $WCET(\tau)$ to be a safe upper bound for the *worst-case execution time (WCET)* of τ on an MPPA tile, in isolation. Note that the WCET values we require are for *parallel* code running on all the 16 processors of a tile. Tight WCET bounds for such code can be computed using the analysis technique proposed in [38].

For each data type t associated with a dependency (simple or delayed), we define the worst-case memory footprint of a value of type t. This information allows the computation of the *worst-case communication time (WCCT)* for a data of that type, using the formula of Section IV-C.

Allocation constraints specify on which tiles a given dataflow block can be executed. In our example, they force the allocation of the capture and display functions onto specific MPPA tiles. More generally, they can be used to confine an application to part of the MPPA, leaving the other tiles free to execute other applications.

VI. Off-line mapping and code generation

A. The problem

The real-time mapping and code generation problem we consider in this paper is a bi-criteria optimization problem: Given a data-flow graph and a non-functional specification, synthesize a real-time implementation that minimizes execution cycle latency (duration) and maximizes throughput[2], with priority given to latency. We chose this scheduling problem because it is meaningful in embedded systems design and because its simple definition allows us to focus on the handling of NoC-related issues. Variations of our mapping algorithms can also handle periodicity, start date, and deadline requirements [13].

Our allocation and scheduling problem being NP-complete, we do not aim for optimality. Instead, we rely on low-complexity heuristics that allow us to handle large numbers of resources with high temporal precision. Mapping and code generation is realized in 3 steps: The first step produces a latency-optimizing scheduling table using the algorithms of Section VI-B. The second step uses the software pipelining algorithms of [12] (not detailed here) to improve throughput while not changing latency. Finally, once a scheduling table is computed, it is implemented it in a way that preserves its real-time properties, as explained in Section VI-C

[2]Throughput in this context means the number of execution cycles started per time unit. It can be different from the inverse of the latency because we allow one cycle to start before the end of previous ones, provided that data-flow dependencies are satisfied.

B. Latency-optimizing scheduling routine

Our scheduling routine builds a *global scheduling table covering all MPPA resources*. It uses a non-preemptive scheduling model for the tasks, and a preemptive one for the NoC communications.[3] For each task it reserves exactly one time interval on one of the tiles. For every dependency between two tasks allocated on different tiles, the scheduling routine reserves one or more time intervals on each resource along the route between the two tiles, starting with the DMA of the source tile, and continuing with the NoC multiplexers (recall that the route is fixed under the X-first routing policy).

The scheduling algorithm uses a simple list scheduling heuristic. The tasks of the dependent task system are traversed one by one in an order compatible with the dependencies between them (only the simple dependencies, not the delayed ones, which are considered during throughput optimization). Resource allocation for a task *and for all communications associated with its input dependencies* is performed upon traversal of the task, and never changed afterwards. Scheduling starts with an empty scheduling table which is incrementally filled as the tasks and the associated communications are reserved time intervals on the various resources.

For each task, scheduling is attempted on all the tiles that can execute the block (as specified by the allocation constraints), at the earliest date possible. Among the possible allocations, we retain the one that minimizes a cost function. This cost function should be chosen so that the final length of the scheduling table is minimized (this length gives the execution cycle latency). Our choice of cost function combines the end date of the task in the schedule (with 95% weight) and the maximum occupation of a CPU in the current scheduling table (with 5% weight). Experience showed that this function produces shorter scheduling tables than the cost function based on task end date alone (as used in [24]).

1) Mapping NoC communications: The most delicate part of our scheduling routine is the communication mapping function **MapCommunicationOnPath**. When a task is mapped on a tile, this function is called once for each of the input dependencies of the task, if the dependency source is on another tile and if the associated data has not already been transmitted.

Fig. 6 presents a (partial) scheduling table produced by our mapping routine. We shall use this example to give a better intuition on the functioning of our algorithms. We assume here that the execution of task f produces data x which will be used by task g. Our scheduling table shows the result of the mapping of task g onto *Tile*$(2,2)$ (which also requires the mapping of the transmission of x) under the assumption that all other tasks (f, h) and data transmissions (y, z, u) were already mapped as pictured (reservations made during the mapping of g have a lighter color).

[3]Task preemptions would introduce important temporal imprecision (through the use of interrupts), and are avoided. Data communications over the NoC are naturally divided into packets that are individually scheduled by the NoC multiplexer programs, allowing a form of pre-computed preemption.

Procedure 1 MapCommunicationOnPath

Input: *Path* : list of resources (the communication path)
StartDate : date after which the data can be sent
DataSize : worst-case data size (in 32-bit words)
Input/Output: *SchedulingTable* : scheduling table

1: **for** i := 1 to ***length***(*Path*) **do**
2: *ShiftSize* := $(i-1) *$ *SegmentBufferSize*;
3: *FreeIntervalList*[i] := ***GetIntervalList***(*SchedulingTable*, ***GetSegment***(*Path*,i), *ShiftSize*)
4: *ShiftedIntervalList*[i] := ***ShiftLeftIntervals***(*FreeIntervalList*[i],*ShiftSize*)
5: **end for**
6: *PathFreeIntervalList* :=***IntersectIntervals***(*ShiftedIntervalList*);
7: (*ReservedIntervals*,*NewIntervalList*,*NewScheduleLength*) := ***ReserveIntervals***(*DataSize*,*PathFreeIntervalList*, ***length***(*SchedulingTable*));
8: (*IntervalForLock*,*NewIntervalList*,*NewScheduleLength*) := ***ReserveIntervals***(*LockPacketLength*,*NewIntervalList*, *NewScheduleLength*);
9: *ReservedIntervals* := ***AppendList***(*ReservedIntervals*,*IntervalForLock*)
10: **for** i := 1 to ***length***(*Path*) **do**
11: *ShiftSize* := (i-1)*SegmentBufferSize*;
12: *FinalIntervals*[i] :=***ShiftRightIntervals***(*ReservedIntervals*,*ShiftSize*);
13: **end for**
14: **if** *NewScheduleLength* $>$ ***length***(*SchedulingTable*) **then**
15: *SchedulingTable* := ***IncreaseLength***(*SchedulingTable*,*NewScheduleLength*);
16: **end if**
17: *SchedulingTable* := ***UpdateSchedulingTable***(*SchedulingTable*,*Path*,*FinalIntervals*);

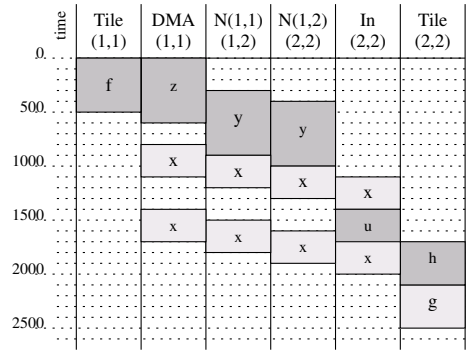

Fig. 6. Scheduling table covering one communication path on our NoC. Only the 6 resources of interest are represented (out of 70). Time flows from top to bottom.

As part of the mapping of g onto *Tile*$(2,2)$, function **MapCommunicationOnPath** is called to perform the mapping of the communication of x from *Tile*$(1,1)$ to *Tile*$(2,2)$. The parameters of its call are *Path*, *StartDate*, and *DataSize*. Parameter *Path* is the list formed of resources *DMA*$(1,1)$, *N*$(1,1)(1,2)$, *N*$(1,2)(2,2)$, and *In*$(2,2)$ (the transmission route of x under the X-first routing protocol). Parameter *StartDate* is given by the end date of task f (in our case 500), and *DataSize* is the worst-case size of the data associated with the data dependency (in our case 500 32-bit words). Time is measured in clock cycles.

To minimize the overall time reserved for a data transmission, we shall require that it is never blocked waiting for a NoC resource. For instance, if the communication of x starts on the *N*$(1,1)(1,2)$ at date t, then on *N*$(1,2)(2,2)$ it must start at date $t +$ *SegmentBufferSize*, where *SegmentBufferSize* is a

platform constant defining the time needed for a flit to traverse one NoC resource. In our NoC this constant is 3 clock cycles (in Fig. 6 we use a far larger value of 100 cycles, for clarity).

Building such synchronized rezervation patterns along the communication routes is what function **MapCommunicationOnPath** does. It starts by obtaining the lists of free time intervals of each resource along the communication path, and realigning them by subtracting $(i-1)*$*SegmentBufferSize* from the start dates of all the free intervals of the i^{th} resource, for all i. Once this realignment is done on each resource by function ***ShiftLeftIntervals***, finding a reservation along the communication path amounts to finding time intervals that are unused on all resources. To do this, we start by performing (in line 6 of function **MapCommunicationOnPath**) an intersection operation returning all realigned time intervals that are free on all resources. In Fig. 6, this intersection operation produces (prior to the mapping of x) the intervals [800,1100) and [1400,2100]. The value 2100 corresponds here to the length of the scheduling table prior to the mapping of g.

We then call function ***ReserveIntervals*** twice, to make reservations for the data transmission and for the lock command associated with each communication. These two functions produce a list of reserved intervals, which then need to be realigned on each resource. In Fig. 6, these 2 calls reserve the intervals [800,1100), [1400,1700), and [1700,1704). The first 2 intervals are needed for the data transmission, and the third is used for the lock command packet.

2) Multiple reservations: Communications are reserved at the earliest possible date, and function ***ReserveIntervals*** allows the fragmentation of a data transmission to allow a better use of NoC resources. In our example, fragmentation allows us to transmit part of x before the reservation for u. If fragmentation were not possible, the transmission of x should be started later, thus delaying the start of g, potentially lengthening the reservation table.

Fragmentation is subject to restrictions arising from the way communications are packetized. More precisely, an interval cannot be reserved unless it has a minimal size, allowing the transmission of at least a packet containing some payload data.

Function **ReserveIntervals** performs the complex translation from data sizes to needed packets and intervals reservations. We present here an unoptimized version that facilitates understanding. This version reserves one packet at a time, using a free interval as soon as it has the needed minimal size. Packets are reserved until the required *DataSize* is covered. Like for tasks, reservations are made as early as possible. For each packet reservation the cost of NoC control (under the form of the *PacketHeaderSize*) must be taken into account.

When the current scheduling table does not allow the mapping of a data communication, function **ReserveIntervals** will lengthen it so that mapping is possible.

Procedure 2 ReserveIntervals

Input: *DataSize* : worst-case size of data to transmit
FreeIntervalList : list of free intervals before reservation
ScheduleLength : schedule length before reservation
Output: *ReservedIntervalList* : reserved intervals
NewIntervalList : list of free intervals after reservation
NewScheduleLength : schedule length after reservation

```
 1: NewIntervalList := FreeIntervalList
 2: ReservedIntervalList := ∅
 3: while DataSize > 0 and NewIntervalList≠ ∅ do
 4:    ival := GetFirstInterval(NewIntervalList);
 5:    NewIntervalList := RemoveFirstInterval(NewIntervalList);
 6:    if IntervalEnd(ival)==ScheduleLength then
 7:       RemainingIvalLength := ∞; /*ival can be extended*/
 8:    else
 9:       RemainingIvalLength := length(ival);
10:    end if
11:    ReservedLength := 0;
12:    while RemainingIvalLength > MinPacketSize and DataSize > 0 do
13:       /*Reserve a packet (clear, but suboptimal code)*/
14:       PacketLength := min(DataSize + PacketHeaderSize,
                              RemainingIvalLength,MaxPacketSize);
15:       RemainingIvalLength -= PacketLength;
16:       DataSize -= PacketLength - PacketHeaderSize;
17:       ReservedLength += PacketLength
18:    end while
19:    ReservedInterval :=
          CreateInterval(start(ival), ReservedLength);
20:    ReservedIntervalList :=
          AppendToList(ReservedIntervalList,ReservedInterval);
21:    if length(ival) - ReservedLength > MinPacketLength then
22:       NewIntervalList := InsertInList(NewIntervalList,
                CreateInterval(start(ival)+ReservedLength,
                               length(ival)-ReservedLength));
23:    end if
24:    NewScheduleLength := max(ScheduleLength,end(ival));
25: end while
```

C. Automatic code generation

Once the scheduling table has been computed, executable code is automatically generated as follows: One sequential execution thread is generated for each tile and for each NoC multiplexer (resources $Tile(i,j)$, $N(i,j)(k,l)$, and $In(i,j)$ in our platform model of Section IV-B). The code of each thread is an infinite loop that executes the (computation or communication) operations scheduled on the associated resource in the order prescribed by their reservations. Recall that each tile contains 16 processor cores, but is reserved as a single sequential resource, parallelism being hidden inside the data-flow blocks. The sequential thread of a tile runs on the first processor core of the tile, but the code of each task can use all 16 processor cores. The code of the NoC multiplexers is executed on the router controllers.

No separate thread is generated for the DMA resource of a tile. Instead, its operations are initiated by thread of the tile. This is possible because the DMA allows the queuing of DMA commands and because mapping is performed so that activation conditions for DMA operations can be computed by the tile resource at the end of data-flow operations. For instance, in the example of Fig. 6, if no other operations are allocated on $Tile(0,0)$, the two DMA operations needed to send x are queued at the end of f.

The synchronization of the threads is realized by explicit lock manipulations by the processors and by the NoC control programs, which force message passing order and implicitly synchronize with the flow of passing messages. The resulting programs enforce the computed *order* of operations on each

resource in the scheduling table, but allow for some timing elasticity: If the execution of an operation takes less than its WCET or WCCT, operations depending on it may start earlier. This elasticity does not compromise the worst-case timing guarantees computed by the mapping tool.

1) Memory handling: Our real-time scheduling and timing analysis use conservative WCET estimations for the (parallel) execution of data-flow blocks on the computing tiles, *in isolation*. Uncontrolled memory accesses coming from other tiles during execution could introduce supplementary delays that are not taken into account in the WCET figures or by our scheduling tool.

To ensure the timing correctness of our real-time scheduling technique, we need to ensure that memory accesses coming from outside a tile do not interfere with memory accesses due to the execution of code inside the tile. This is done by exploiting the presence of multiple memory banks on each tile. The basic idea is to ensure that incoming DMA transfers never use the same memory banks as the code running at the same time on the CPUs. Of course, once a DMA transfer is completed, the memory banks it has modified can be used by the CPUs, the synchronization being ensured through the use of locks.

We currently ensure this property at code generation time, by explicitly allocating variables to memory banks in such a way as to exclude contentions. While not general, this technique worked well for our case studies, which allowed us to focus the work of this paper on dealing with the NoC resource allocation problem. We are currently working on integrating RAM bank allocation within the mapping algorithm.

VII. Evaluation

We have evaluated our mapping and code generation method on two applications: The platooning application described in Section V and a parallel Cooley-Tukey implementation of the integer 1D radix 2 FFT over 2^{14} samples [6]. We chose these two applications because they allow the computation of tight lower bounds on the execution cycle latency and because (for the FFT) an MPPA mapping already exists. This allows for meaningful comparisons, while no tool equivalent to ours was available for evaluation.

For the FFT, we followed the parallelization scheme used in [6], with a block size of 2^{11}, resulting in a total of 32 tasks. Evaluation is done on the 3x4 MPPA pictured in Fig. 2, where we assume that input data arrives on *Tile*$(0,0)$ and the results are output by *Tile*$(2,3)$.

For both applications, after computing the WCET of the tasks and the WCCT of the data transmissions, the mapping tool was applied to build a running implementation and to compute execution cycle latency and throughput guarantees. Then, the code was run, and its performances measured. This allowed us to check the functional correctness of the code and to determine that our tool produces very *precise timing guarantees*. Indeed, the difference between predicted and observed latency and throughput figures is less than 1% for both examples, which is due to the precision of our mapping algorithms and to the choice of a very predictable execution platform.

The generated off-line schedule (and the resulting code) has *good real-time properties*. For both the CyCab and the FFT, we have manually computed lower bounds on the execution cycle latency.[4] The lower bounds computed for the CyCab and FFT examples were lower than the latency values computed by our algorithms by respectively 8.9% and 3.4%. For the FFT example, we have also compared the measured latency of our code with that of a classical NoC-based parallel implementation of the FFT [6] running on our architecture. For our code, the NoC was statically scheduled, while for the classical implementation it was not. Execution results show that our code had a latency that was 3.82% shorter than the one of the classical parallel FFT code. In other words, **our tool produced code that not only has statically-computed hard real-time bounds (which the hand-written code has not) but is also faster**.

Our mapping heuristics favor the concentration of all computations and communications in a few tiles, leaving the others free to execute other applications (as opposed to evenly spreading the application tasks over the tiles). The code generated for Cycab has a tile load of 85%-99% for 6 of the 12 tiles of the architecture, while the other tiles are either unused or with very small loads (less than 7%). Using more computing tiles would bring no latency or throughput gains because our application is limited by the input acquisition speed. In the FFT application the synchronization barriers reduce average tile use to 47% on 8 of the 12 MPPA tiles. Note that the remaining free processor and NoC time can be used by other applications.

Finally, we have measured the influence of static scheduling of NoC communications on the application latency, by executing the code generated for Cycab and the FFT with and without NoC programming. For Cycab, not programming the NoC results in a speed loss of 7.41%. For the FFT the figure is 4.62%.

We conclude that our tool produces global static schedules of good quality, which provide timing guarantees close to the optimum.

VIII. Conclusion

We have defined a new technique and tool for the off-line real-time mapping of data-flow graphs onto many-core architectures supporting programmed arbitration of NoC communications. In doing this, we have shown that *taking into account the fine detail of the hardware and software architecture allows off-line mapping of very good precision*, even when low-complexity scheduling heuristics are used in order to ensure the scalability of the approach. Our tool synthesizes

[4]To compute these lower bounds we simplify the hardware model by assuming that the resources $N(i,j)(k,l)$ generate no contention (*i.e.* they allow the simultaneous transmission of all packets that demand it). We only take into account the sequencing of operations on processors and DMAs and the contentions on resources $In(i,j)$.

code with static real-time guarantees that runs faster than (simple) hand-written parallel code.

For the future, our first objective is to perform memory resource allocation at scheduling time (and not during code generation). More generally, our mapping technique could benefit from/to previous work on the scheduling of data-flow specifications and on compilation, but complex evaluation is needed to determine which algorithms scale up to take into account the low-level architectural detail.

REFERENCES

[1] "The Epiphany many-core architecture." www.adapteva.com, 2012.

[2] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, no. 2, Jun. 2005.

[3] C. H. and, "Pnoc: a flexible circuit-switched noc for fpga-based systems," *IEE Proceedings on Computers and Digital Techniques*, vol. 153, no. 3, 2006.

[4] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey, "Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor," in *Proceedings ALCHEMY 2013*, Barcelona, Spain, June 2013.

[5] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, "Methods and tools for mapping process networks onto multi-processor systems-on-chip," in *Handbook of Signal Processing Systems*. Springer, 2013.

[6] J. H. Bahn, J. Yang, and N. Bagherzadeh, "Parallel FFT algorithms on network-on-chips," in *Proceedings ITNG 2008*, april 2008.

[7] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "A framework to schedule parametric dataflow applications on many-core platforms," in *Proceedings CPC'13*, Lyon, France, 2013.

[8] S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*. Springer, 2013, 2nd edition, in particular chapter.

[9] T. Bjerregaard and J. Sparso, "Implementation of guaranteed services in the mango clockless network-on-chip," *Computers and Digital Techniques*, vol. 153, no. 4, 2006.

[10] S. Borkar, "Thousand core chips – a technology perspective," in *Proceedings DAC*, San Diego, CA, USA, 2007.

[11] E. Carara, N. Calazans, and F. Moraes, "Router architecture for high-performance nocs," in *Proceedings SBCCI*, Rio de Janeio, Brazil, 2007.

[12] T. Carle and D. Potop-Butucaru, "Throughput Optimization by Software Pipelining of Conditional Reservation tables," INRIA, Rapport de recherche RR-7606, Apr. 2011, to appear in ACM TACO. [Online]. Available: http://hal.inria.fr/inria-00587319

[13] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens, "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation," INRIA, Research report RR-8109, Oct. 2012. [Online]. Available: http://hal.inria.fr/hal-00742908

[14] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, Oct. 2011.

[15] M. Djemal, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang, "Programmable routers for efficient mapping of applications onto NoC-based MPSoCs," in *Proceedings DASIP*, Karlsruhe, Germany, 2012.

[16] P. Eles, A. Doboli, P. Pop, and Z. Peng, "Scheduling with bus access optimization for distributed embedded systems," *IEEE Transactions on VLSI Systems*, vol. 8, no. 5, pp. 472–491, Oct 2000.

[17] J. H. et al., "A 48-core ia-32 processor in 45nm cmos using on-die message-passing and dvfs for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, Jan 2011.

[18] E. W. *et al.*, "Baring it all to software: The raw machine," *IEEE Computer*, vol. 30, no. 9, pp. 86 –93, sep 1997.

[19] G. Fohler and K. Ramamritham, "Static scheduling of pipelined periodic tasks in distributed real-time systems," in *In Procs. of EUROMICRO-RTS97*, 1995, pp. 128–135.

[20] D. Genius, A. M. Kordon, and K. Z. el Abidine, "Space optimal solution for data reordering in streaming applications on noc based mpsoc," *Journal of System Architecture*, 2013.

[21] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat, "Time analysable synchronisation techniques for parallelised hard real-time applications," in *Proceedings DATE'12*, Dresden, Germany, 2012.

[22] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, Sep 2003.

[23] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, 2005.

[24] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings MEMOCODE*, Mont Saint-Michel, France, June 2003.

[25] D. Hardy and I. Puaut, "Wcet analysis of multi-level non-inclusive set-associative instruction caches," in *RTSS*, 2008.

[26] M. Harrand and Y. Durand, "Network on chip with quality of service," United States patent application publication US 2011/026400A1, Feb. 2011.

[27] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programmin*. Morgan Kaufmann, 2008.

[28] H. Kashif, S. Gholamian, R. Pellizzoni, H. Patel, and S. Fischmeister, "Ortap: An offset-based response time analysis for a pipelined communication resource model," in *Proceedings RTAS*, 2013.

[29] C. L. . J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, 1973.

[30] Z. Lu and A. Jantsch, "Tdm virtual-circuit configuration for network-on-chip," *IEEE Trans. VLSI*, 2007.

[31] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network-on-chip," in *Proceedings DATE*, 2004.

[32] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *Proceedings ISCA-36*, 2009.

[33] "The MPPA256 many-core architecture." www.kalray.eu, 2012.

[34] L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, 1993.

[35] B. Nikolic, H. Ali, S. Petters, and L. Pinho, "Are virtual channels the bottleneck of priority-aware wormhole-switched noc-based many-cores?" in *Proceedings RTNS*, 2013, October 2013.

[36] I. M. Panades, A. Greiner, and A. Sheibanyrad, "A low cost network-on-chip with guaranteed service well suited to the GALS approach," in *Proceedings NanoNet'06*, Lausanne, Switzerland, Sep 2006.

[37] C. Pradalier, J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier, "The CyCab: a car-like robot navigating autonomously and safely among pedestrians," *Robotics and Autonomous Systems*, vol. 50, no. 1, 2005.

[38] I. Puaut and D. Potop-Butucaru, "Integrated worst-case execution time estimation of multicore applications," in *Proceedings WCET'13*, Paris, France, July 2013.

[39] A. Racu and L. Indrusiak, "Using genetic algorithms to map hard real-time on noc-based systems," in *Proceedings ReCoSoC*, July 2012.

[40] Z. Shi and A. Burns, "Schedulability analysis and task mapping for real-time on-chip communication," *Real-Time Systems*, vol. 46, no. 3, pp. 360–385, 2010.

[41] R. Sorensen, M. Schoeberl, and J. Sparso, "A light-weight statically scheduled network-on-chip," in *Proceedings NORCHIP*, 2012.

[42] "The TilePro64 many-core architecture." www.tilera.com, 2008.

[43] C. Villalpando, A. Johnson, R. Some, J. Oberlin, and S. Goldberg, "Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander," in *Proceedings of the IEEE Aerospace Conference*, 2010.

[44] R. Wilhelm and J. Reineke, "Embedded systems: Many cores – many problems (invited paper)," in *Proceedings SIES'12*, Karlsruhe, Germany, June 2012.

[45] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *Software Engineering, IEEE Transactions on*, vol. 19, no. 2, pp. 139–154, 1993.

[46] Y. Yoon, N. Concer, M. Petracca, and L. Carloni, "Virtual channels vs. multiple physical networks: a comparative analysis," in *Proceedings DAC*, Anaheim, CA, USA, 2010.

[47] J. T. Zhai, M. Bamakhrama, and T. Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *Proceedings DAC*, 2013.